UCRL-ID-123220

UniTree Name Server Internals

D. Mecozzi
J. Minton

January 1996

Lawrence Livermore National Laboratory

This is an informal report intended primarily for internal or limited external distribution. The opinions and conclusions stated are those of the author and may or may not be those of the Laboratory.
Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This report has been reproduced
directly from the best available copy.

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information
P.O. Box 62, Oak Ridge, TN 37831
Prices available from (615) 576-8401, FTS 626-8401

Available to the public from the
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd.,
Springfield, VA 22161

UniTree Name Server Internals[*]

Donna Mecozzi and Jim Minton
Lawrence Livermore National Laboratory
Livermore, California

INTRODUCTION

The UniTree Name Server (UNS) is one of several servers which make up the UniTree storage system. The Name Server is responsible for mapping *names* to *capabilities*. *Names* are generally human readable ASCII strings of any length. *Capabilities* are unique 256-bit identifiers that point to files, directories, or symbolic links. The Name Server implements a UNIX style hierarchical directory structure to facilitate name-to-capability mapping. The principal task of the Name Server is to manage the directories which make up the UniTree directory structure.

The principle clients of the Name Server are the FTP Daemon, NFS and a few UniTree utility routines. However, the Name Server is a generalized server and will accept messages from any client.

The purpose of this paper is to describe the internal workings of the UniTree Name Server. In cases where it seems appropriate, the motivation for a particular choice of algorithm as well as a description of the algorithm itself will be given.

1.0 Directories

Each UniTree directory contains state information and an arbitrary number of *entries* . An entry is a *name-capability pair*. An example of a piece of state information is the directory owners UID. The Name Server implements directories by chaining together one or more (1024 byte) blocks from a file managed by Cachelib. (The Cachelib software layer is shown in Figure 12 and is discussed in section 3.4.)

Figure 1 illustrates a directory composed of several Cachelib blocks chained together. The first block in every directory is called a BlockHead. When the entry space in the BlockHead becomes full, an EntryBlock is attached to the BlockHead. When the entry space in an EntryBlock becomes full, another EntryBlock is attached to the last EntryBlock in the directory's EntryBlock chain. Thus, a directory can grow to accommodate an arbitrary number of entries. A directory may also have one or more NameBlocks, as shown in Figure 1. NameBlocks hold "overflow" characters from long entry names and very long symbolic link text. As with EntryBlocks, additional NameBlocks are attached to the last NameBlock in the directory 's NameBlock chain when the required name space is unavailable.

[*] This work was performed under the auspices of the U. S. Department of Energy by Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48.

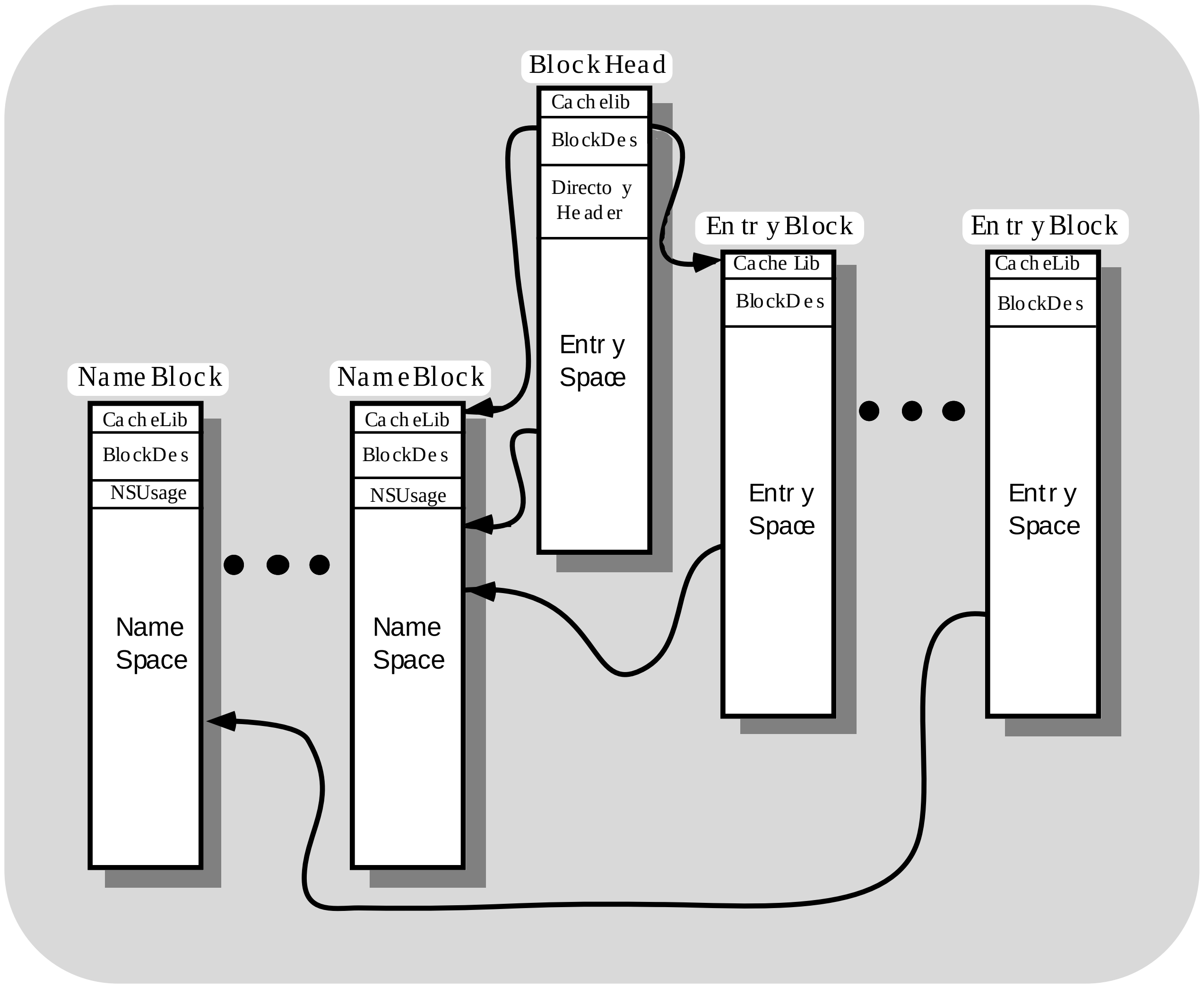

Figure 1. This drawing illustrates the composition of a UniTree directory. A directory consists of one or more 1024 byte blocks. A directory always has one and only one BlockHead. A directory has zero or more NameBlocks and zero or more Entry Blocks.

1.1 BlockHead

The BlockHead is the “top” level block in a directory and is composed of four distinct structures:

• a Cachelib header	(16 bytes of data)
• a BlockDescriptor	(32 bytes of data)
• a DirectoryHeader	(112 bytes of data)
• an EntrySpace	(864 bytes)

Figure 2 shows an overview of a BlockHead.

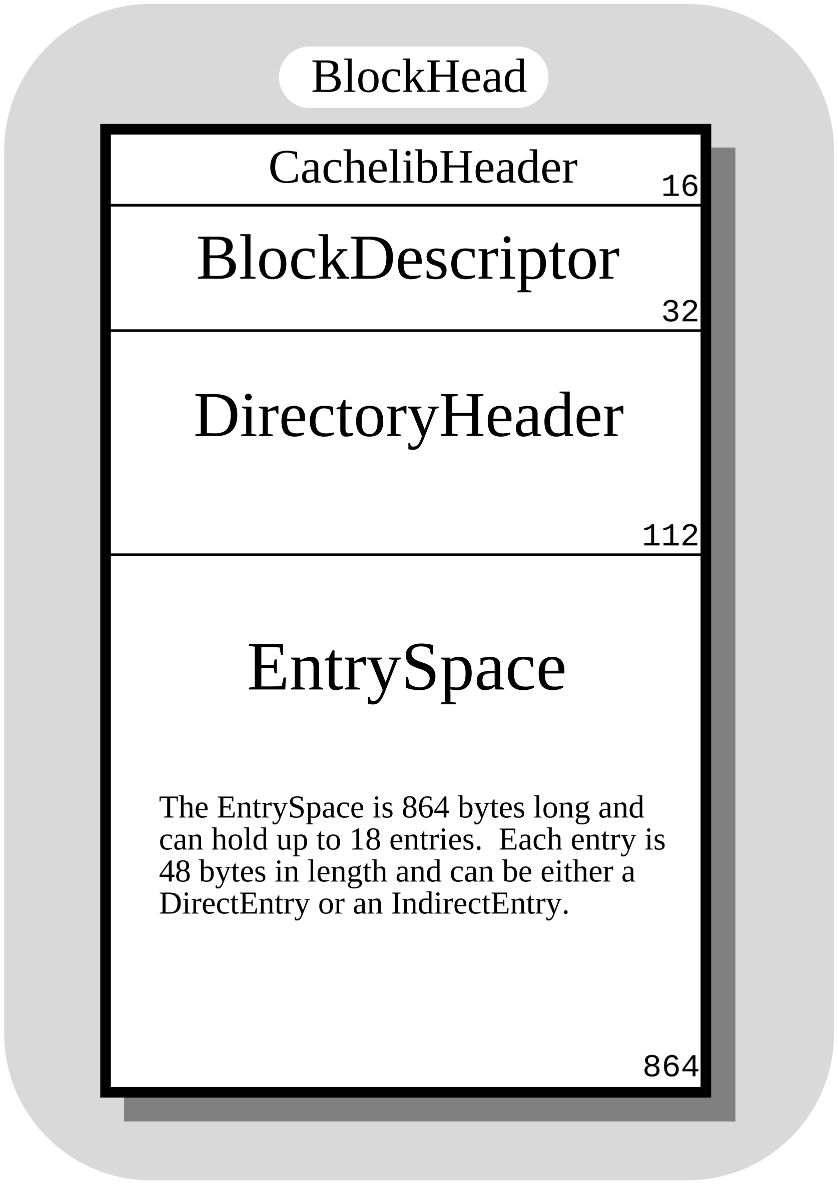
BlockHead
CachelibHeader
16
BlockDescriptor
32
DirectoryHeader
112
EntrySpace
The EntrySpace is 864 bytes long and can hold up to 18 entries. Each entry is 48 bytes in length and can be either a DirectEntry or an IndirectEntry.
864

Figure 2. The BlockHead Structure is a composite of four structures: a CachelibHeader, a BlockDescriptor, a DirectoryHeader, and an EntrySpace.

In include file `nsdir.h` a BlockHead is defined as:

```
typedef struct
{
    BlockDescriptorStruct  BlockDes;
    DirectoryHeaderStruct  DirHeader;
    DirectEntryStrct       Entry[MAX_ENTRIES_IN_BLOCK_HEAD];
} BlockHeadStruct;
```

The CachelibHeader is actually part of the structure of the block obtained from the Cachelib software, so its structure is not defined as part of the BlockHead structure.

1.1.1 Cachelib Header

A Cachelib header has three fields. The FreeList pointer is non zero if the block is on the free (available) list. The Checksum will be non zero if Cachelib has been instructed to checksum the blocks. The TimeStamp contains the time the block was written. A Cachelib header is shown in Figure 3. A discussion of the Cachelib software layer is given below in section 3.4.

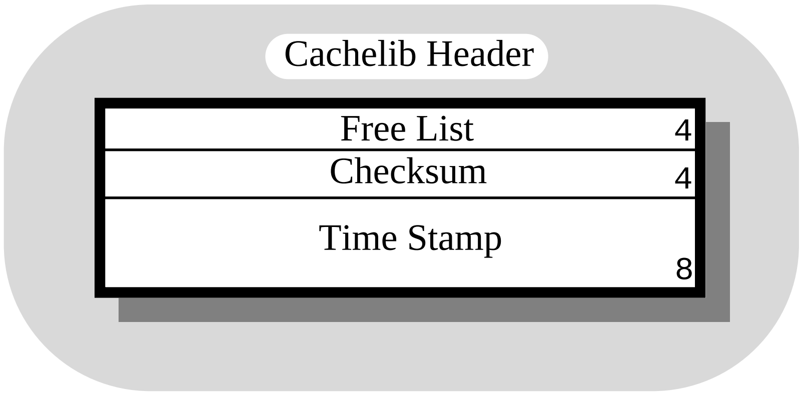

Figure 3. The Cachelib Header occupies four 32-bit words. The field widths in this diagram are given in bytes.

In include file `cache.h` the CachelibHeader is defined as:

```
typedef struct ObjectHeaderStruct
{
   unsigned   FreeList;
   unsigned   Checksum;
   int64      TimeStamp;
} ObjectHeaderStruct;
```

1.1.2 Block Descriptors

The BlockDescriptor contains data describing how the cachelib block is used by the Name Server. Figure 4 shows the format of a BlockDescriptor. The Consistency Word contains one of the following strings, "BkHd", "EnBk", "NmBk" or "SmLk", depending on how the block is being used. The BlockHeadId, NextBlockId and NameBlockId contain Cachelib identifiers. The bit fields (InUse and Indirect) in the BlockDescriptor have a one-to-one mapping to each entry in the EntrySpace. These fields indicate whether or not an entry is currently InUse and whether an InUse entry is an IndirectEntry or not. (The value of a specific InUse bit is 1 whenever the corresponding EntrySpace has an entry in it. The value of a specific Indirect bit is 1 whenever the corresponding entry should be interpreted as an IndirectEntry.) Depending on how the block is being used, one and only one of the BlockHead, EntryBlock, NameBlock or SymbolicLink bits is set. The TrashDirectory bit is set in the BlockHead of a “.trash” directory. (Trash directories are discussed in section 2.0.) Finally, ThisBlockId contains the Cachelib identifier of the block itself.

The ConsistencyWord of a BlockHead contains the string "BkHd". The BlockHead bit will be set true (set to 1). The BlockHeadId and ThisBlockId will be equal and set to the value of the Cachelib identifier of the block. A BlockHead only holds 18 entries, so only the first 18 Indirect/InUse elements will be used by the BlockHead.

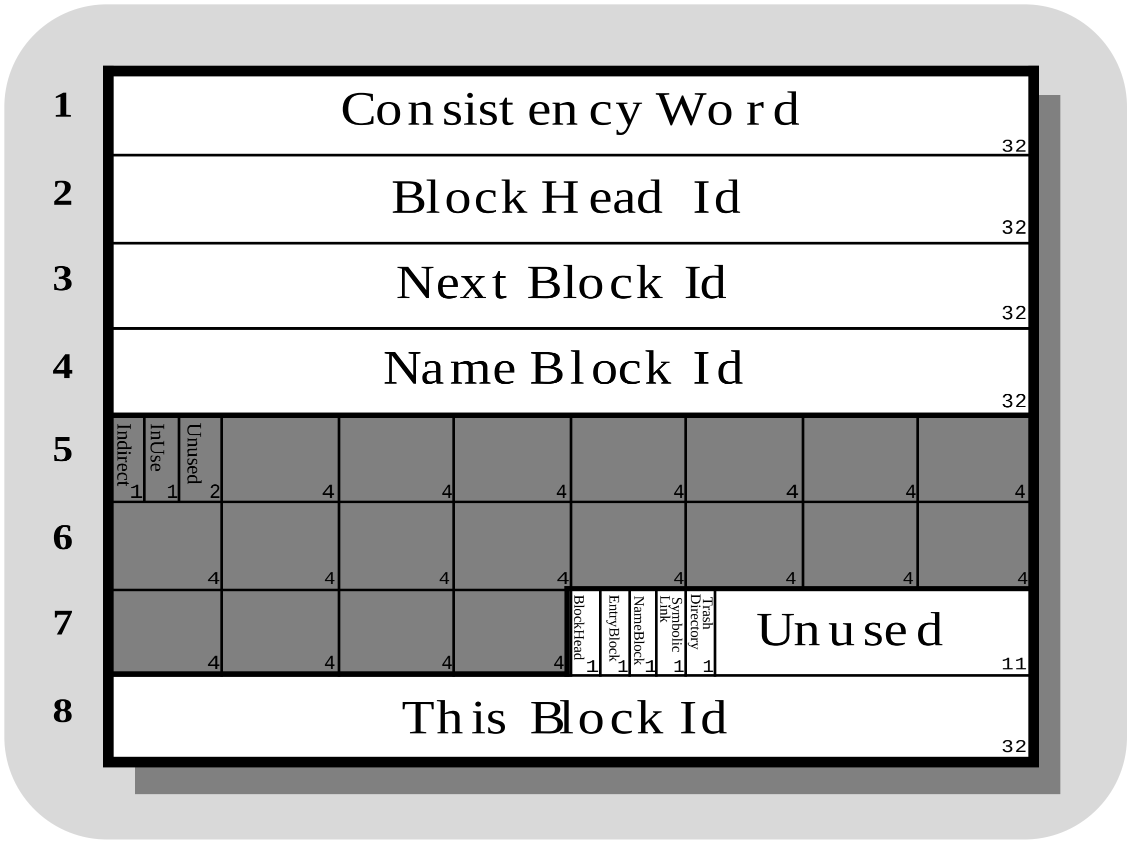

Figure 4. The BlockDescriptor occupies eight 32-bit words. The field widths in this diagram are given in bits.

In include file `nsdir.h` a the BlockDescriptor is defined as:

```
typedef struct
{
   ConsistencyString  ConsistencyWord;
   unsigned           BlockHeadId;
   unsigned           NextBlockId;
   unsigned           NameBlockId;
   unsigned           EntrySpaceMap1;
   unsigned           EntrySpaceMap2;
   unsigned           EntrySpaceMap3    : 16;
   unsigned           BlockHead         :  1;
   unsigned           EntryBlock        :  1;
   unsigned           NameBlock         :  1;
   unsigned           SymbolicLink      :  1;
   unsigned           TrashDirectory    :  1;
   unsigned           Unused1           : 11;
   unsigned           ThisBlockId;
} BlockDescriptorStruct;
```

1.1.3 Directory Header

The Directory Header contains information necessary to provide UNIX style access to the directory. Figure 5 depicts a DirectoryHeader structure. The Directory Header has a field for the UID and GID of the user that owns the directory. The Version field is used to note significant changes in the Name Server's data base. LinkCount contains a count of the number of directories that reference the directory. The four time fields, TimeCreated, TimeLastAccessed, TimeLastModified, and TimeHeaderUpdated are micro second time values that indicate when the directory was created, last read, entry space was last modified, and when data in the Directory Header last changed. The ProtectionLevel is advisory only and ranges in value from 0 to 7. Permissions are the standard UNIX mode bits, read, write and execute (rwx), with mode bits for the owner, group and others. The ExpirationTime is used only for ".trash" directories (discussed in section 2.0). NumBlocks is a count of the number of blocks comprising the directory. The EncryptionKey is a unique value used to decrypt capabilities and verify that they identify the directory.

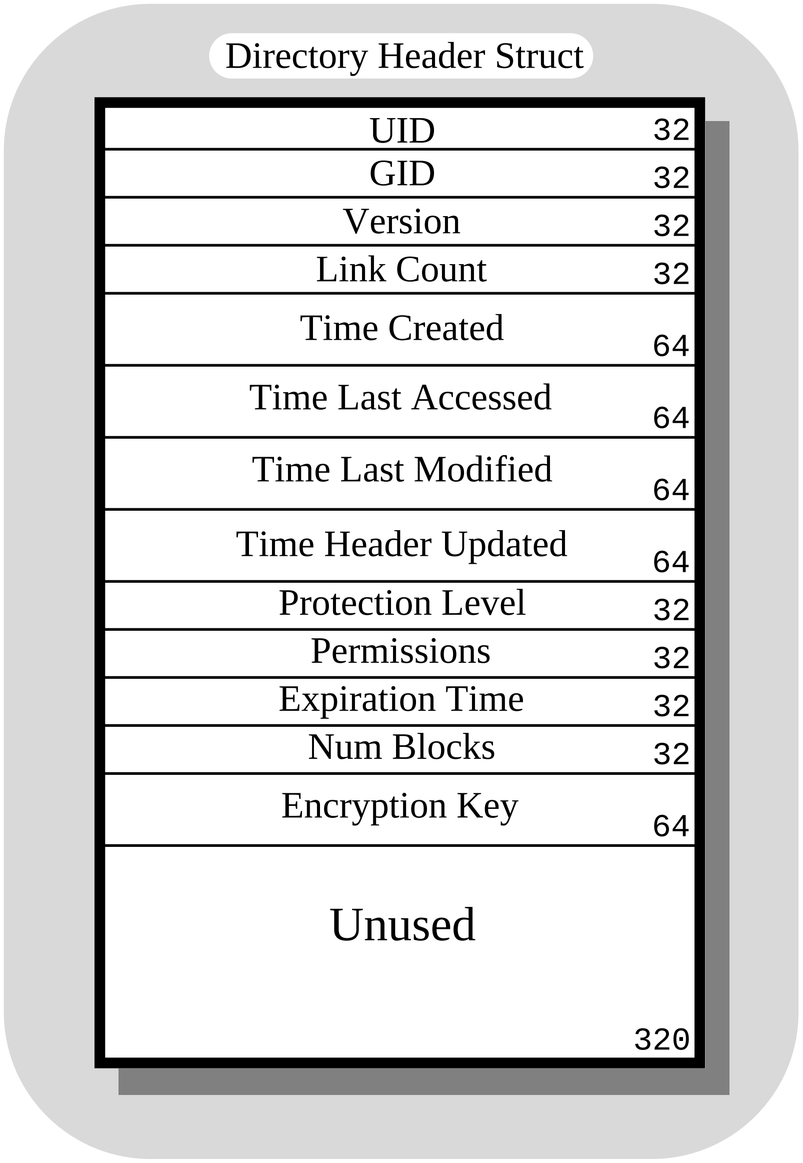

Figure 5. A Directory Header structure. The field widths in this diagram are given in bits.

In include file `nsdir.h` a the Directory Header is defined as:

```
typedef struct
{
    int         UID;
    int         GID;
    unsigned    Version;
    int         LinkCount;
    int64       TimeCreated;
    int64       TimeLastAccessed;
    int64       TimeLastModified;
    int64       TimeHeaderLastModified;
    unsigned    ProtectionLevel;
    unsigned    Permissions;
    int         ExpirationTime;
    unsigned    NumBlocks;
    int64       EncryptionKey;
    unsigned    Unused2[10];
} DirectoryHeaderStruct;
```

1.1.4 EntrySpace

The BlockHead has enough EntrySpace to hold 18 entries. When a new entry is to be inserted into the directory, the Name Server examines the InUse bits to find the first available unused entry in the EntrySpace. The appropriate InUse bit becomes true (set to 1) when the entry is added to the EntrySpace. When an entry is deleted the corresponding InUse bit becomes false and the entry space becomes available for new entries. There are two types of entries: DirectEntries and IndirectEntries.

Direct Entries

DirectEntries are composed of a Name and Capability pair. Figure 6 depicts the structure of a DirectEntry. The length of the name string that can be kept in a DirectEntry is 16 characters or less. When a string is less than 16 characters long, a NIL character is appended to the end of the string. That way, the length of the entry name can easily be determined. An entry name of exactly 16 characters does not have a trailing NIL character.

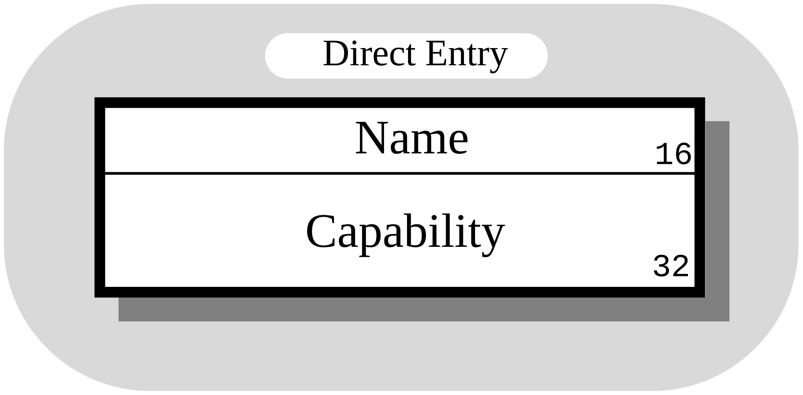

Figure 6. The structure of a Direct Entry. The field widths in this diagram are given in bytes.

In include file `nsdir.h` a the DirectEntry structure is defined as:

```
typedef struct
{
    DirectNameString    Name;
    Capability          ResourceId;
} DirectEntryStruct;
```

Indirect Entries

IndirectEntries are also composed of a Name and Capability pair, but since the Name is greater than 16 characters part of it overflows into a NameBlock. The first 8 characters of the name are kept in the IndirectEntry and the remaining characters are kept in one or more NameBlocks. An IndirectEntry also contains a "pointer" to locate the characters that are stored in a NameBlock. The "pointer" contains two pieces of data: a NameBlockId and an Offset. The NameBlockId is the Cachelib identifier of the NameBlock were the string begins. The Offset is a zero origin index into the NameSpace and points to the beginning of the overflow string.

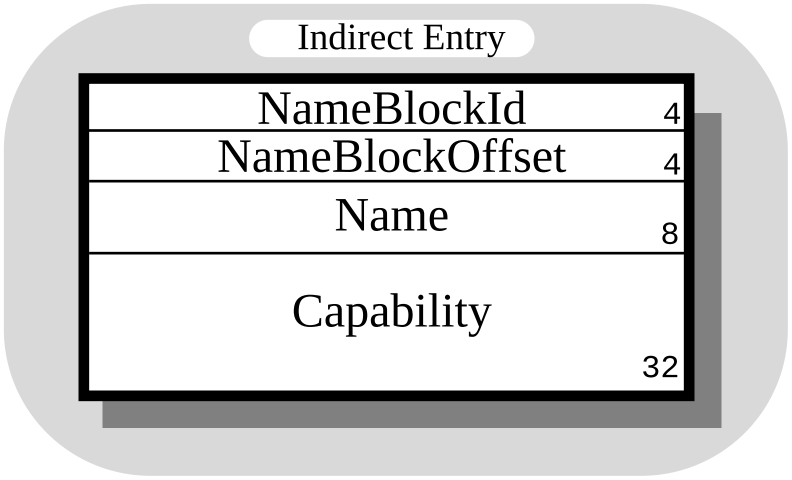

Figure 7. The structure of an IndirectEntry. The field widths in this diagram are given in bytes.

In include file `nsdir.h` a the IndirectEntry structure is defined as:

```
typedef struct
{
    unsigned             NameBlockId;
    unsigned             NameBlockOffset;
    PartialNameString    Name;
    Capability           ResourceId;
} IndirectEntryStruct;
```

1.2 Name Blocks

NameBlocks are used to hold “overflow” characters from entry names that exceed 16 characters and symbolic link text that exceeds 864 characters. A NameBlock is composed of the following fields:

• a Cachelib header	(16 bytes of data)
• a BlockDescriptor	(32 bytes of data)
• a HoleSize	(4 bytes of data)
• a NameSpaceUsage bits	(32 bytes of data)
• Name Space	(940 bytes)

Figure 8 shows an overview of a NameBlock.

The ConsistencyWord in the BlockDescriptor of a NameBlock will be "NmBk". In addition, the NameBlock bit in the BlockDescriptor will be true (set to 1). The Indirect/InUse elements of the BlockDescriptor are not used by a NameBlock and should all be set to 0. The value of the BlockHeadId in the BlockDescriptor will be that of the BlockHead that the NameBlock in associated with.

1.2.1 HoleSize

The HoleSize contains the character length of the largest string that can be placed in the NameBlock. This value is computed each time the name space is altered by the addition or deletion of characters. (The NameSpaceUsage bits are used to compute the HoleSize.) The HoleSize facilitates finding a NameBlock that can accommodate a string.

1.2.2 NameSpaceUsage bits

NameSpaceUsage is an array of bits each of which correspond to a (32-bit) word in the NameSpace. Whenever characters are put into a word of the NameSpace, the corresponding NameSpaceUsage bit is turned on (set to 1). For example, if characters are put into words 193 and 194 of the NameSpace area, bits 193 and 194 in the NameSpaceUsage bit array are turned on. Similarly, when characters are removed from the NameSpace, the corresponding NameSpaceUsage bits are turned off (set to 0).

1.2.3 Name Space

The NameSpace is 235 (32-bit) words of data. The NameSpace is allocated in units of words. The strings stored in the NameSpace are of variable length and are terminated with a NIL character. That way, each string stored in the NameSpace can be easily identified.

When the Name Server is trying to locate a NameBlock in which to store an "overflow" string, it checks the HoleSize of attached NameBlocks. If the HoleSize of a NameBlock is large enough to hold the string plus the trailing NIL character, then the NameSpaceUsage bits of that NameBlock are examined to locate the hole in the NameBlock. If there is no hole large enough to hold the string, or there are no NameBlocks associated with the directory, then a NameBlock is obtained and linked onto the NameBlock chain.

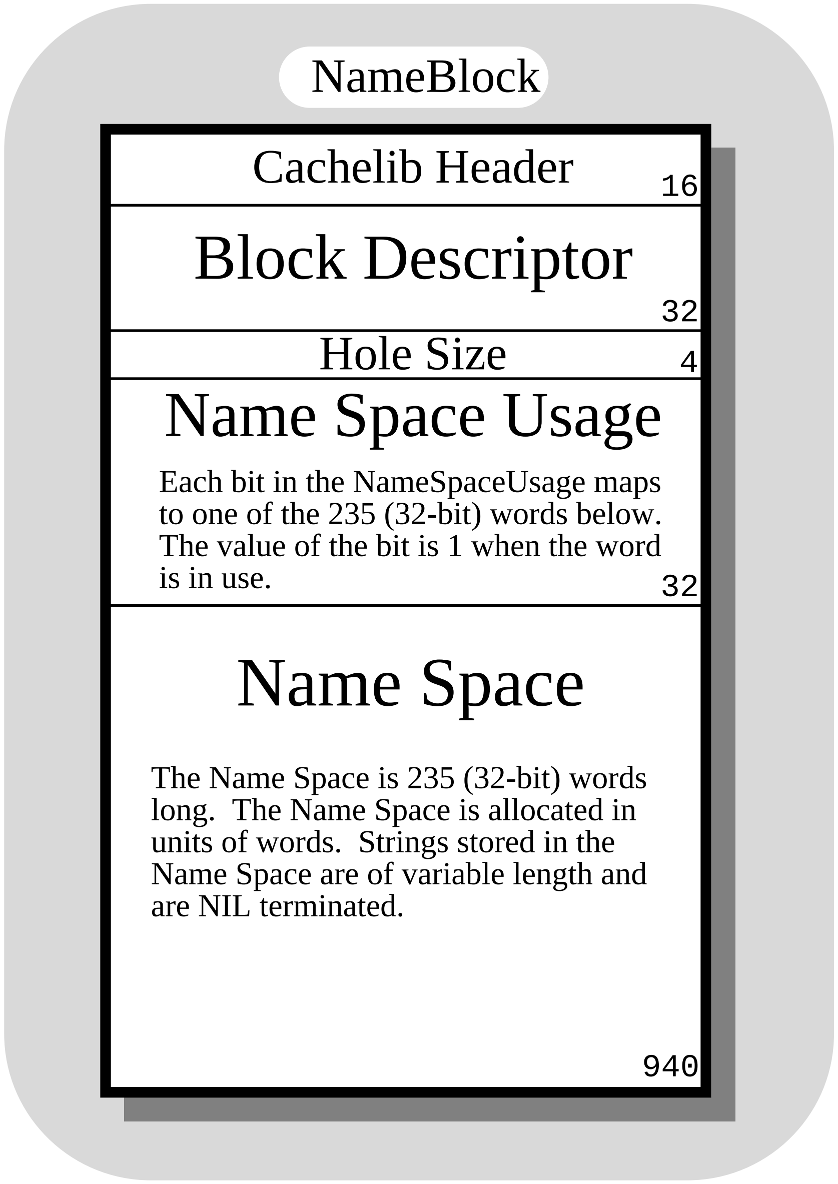
NameBlock
Cachelib Header
16
Block Descriptor
32
Hole Size
4
Name Space Usage
Each bit in the NameSpaceUsage maps to one of the 235 (32-bit) words below. The value of the bit is 1 when the word is in use.
32
Name Space
The Name Space is 235 (32-bit) words long. The Name Space is allocated in units of words. Strings stored in the Name Space are of variable length and are NIL terminated.
940

Figure 8. NameBlocks hold “overflow” characters from strings that are too long to fit into other blocks. Field widths are given in bytes.

In include file `nsdir.h` a NameBlock is defined as:

```
typedef struct
{
    BlockDescriptorStruct  BlockDes;
    unsigned          HoleSize;
    unsigned          NameSpaceUsage[NAME_SPACE_USAGE_LENGTH];
    char              Name[NAME_SPACE_CHAR_LENGTH];
} BlockHeadStruct;
```

The CachelibHeader is actually part of the structure of the block obtained from the cachelib software, so its structure is not defined as part of the NameBlock structure.

1.2.4 Storing Long Names

A *first-fit* algorithm is used to find a place for overflow characters. If there are no NameBlocks attached to the directory, one is obtained and linked to the BlockHead. If one or more NameBlocks are attached to the directory, each NameBlock in the chain is examined, in the order they are linked, for a hole large enough to hold the characters. The hole must be large enough to hold the characters plus a trailing NIL character. If a hole large enough is found, the characters are copied into the hole, the NameBlock is marked "dirty" and all of the blocks associated with this update (transaction) are written to the database.

If a large enough hole cannot be found in any of the NameBlocks a new NameBlock is obtained. It is linked to the last NameBlock in the chain, and the characters are put into this new NameBlock. If a string contains more characters than will fit in a single empty NameBlock, as many NameBlocks as needed are obtained and linked to the directory, until all of the characters have been put into NameBlocks. Whenever the last character of a NameBlock is non-NIL, the NameBlock is being used to hold a string that is greater-than or equal-to 864 characters in length.

There is an interesting end-case to note. Suppose a string that is a multiple of 864 characters is being stored into a NameBlock. Such a string will exactly fill a (multiple of) NameBlock(s). But there must be space for the trailing NIL character. The NIL character will be placed into the first character position of the last (newly obtained) NameBlock.

1.3 EntryBlocks

EntryBlocks are used to hold entries that exceed the 18 entries that can be kept in a BlockHead. The first EntryBlock that is attached to a BlockHead will be "pointed at " by the NextBlockId field in the BlockHead's BlockDescriptor. As additional EntryBlocks are added to the directory, they are serially linked through the NextBlockId field in the BlockDescriptor.

An EntryBlock is composed of the following fields:

• a Cachelib header	(16 bytes of data)
• a BlockDescriptor	(32 bytes of data)
• Entry Space	(960 bytes)

Figure 9 shows an overview of an EntryBlock.

The ConsistencyWord in the BlockDescriptor of an EntryBlock will be "EnBk". In addition, the EntryBlock bit in the BlockDescriptor will be true (set to 1). An EntryBlock can hold up to 20 entries so all of the Indirect/InUse elements are used by an EntryBlock. The value of the BlockHeadId in the BlockDescriptor will be the Id of the BlockHead that the EntryBlock is associated with.

Like the BlockHead's EntrySpace, an EntryBlock's EntrySpace is used to store Direct and Indirect entries. Entry types are discussed in section 1.1.4.

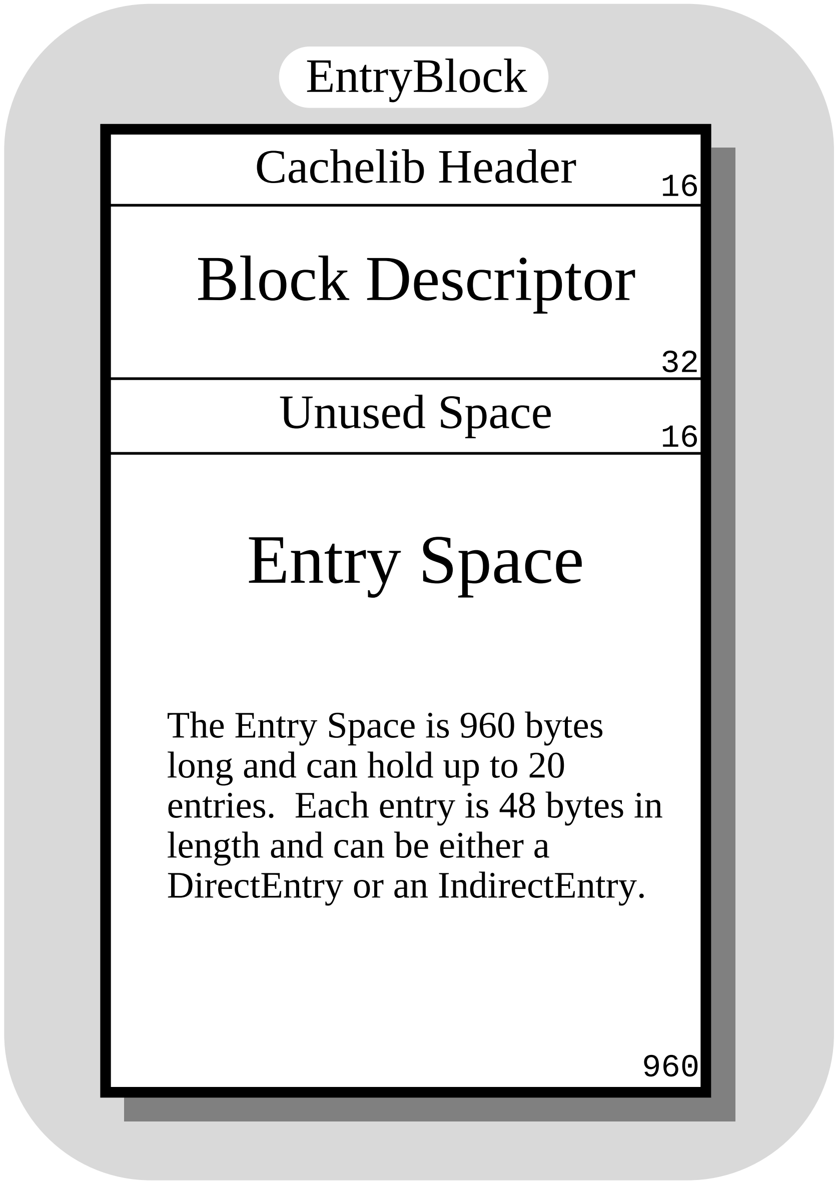
EntryBlock
Cachelib Header
16
Block Descriptor
32
Unused Space
16
Entry Space
The Entry Space is 960 bytes long and can hold up to 20 entries. Each entry is 48 bytes in length and can be either a DirectEntry or an IndirectEntry.
960

Figure 9. An overview of an EntryBlock. The field widths in this diagram are given in units of bytes.

In include file `nsdir.h` a an EntryBlock is defined as:

```
typedef struct
{
    BlockDescriptorStruct  BlockDes;
    unsigned               Unused[4];
    DirectEntryStruct      Entry[MAX_ENTRIES_IN_ENTRY_BLOCK];
} BlockHeadStruct;
```

The CachelibHeader is actually part of the structure of the block obtained from the cachelib software, so its structure is not defined as part of the EntryBlock structure.

1.4 Symbolic Link Blocks

The SymbolicLinkBlock is the “top” level block in a symbolic link and is composed of four distinct structures:

• a Cachelib header	(16 bytes of data)
• a BlockDescriptor	(32 bytes of data)
• a DirectoryHeader	(112 bytes of data)
• NameSpace	(864 bytes)

Figure 10 shows an overview of a SymbolicLink.

The ConsistencyWord of the BlockDescriptor in a SymbolicLink contains the string, "SmLk". The SymbolicLink bit in the BlockDescriptor will be true (set to 1). The BlockHeadId field and the ThisBlockId field will be equal and contain the value of the Cachelib identifier of the block. NameBlocks are the only other type of block that will be chained to a SymbolicLink, and they will be chained through the NameBlockId field. The Indirect/InUse elements are not used by a SymbolicLink and should all be set to 0.

The information kept in the DirectoryHeader structure of a SymbolicLink is identical to that kept for a directory. In addition, there are no semantic differences in the way the DirectoryHeader information is interpreted by the Name Server for a SymbolicLink or a directory. The DirectoryHeader is discussed in section 1.1.3.

The NameSpace of a SymbolicLink can hold up to 864 characters. When the data to be kept exceeds 864 characters, one or more NameBlocks is attached to the SymbolicLink through the NameBlockId fields. No distinction is made between the way overflow entry names and overflow SymbolicLink data are stored in NameBlocks. NameBlocks were discussed in section 1.2.

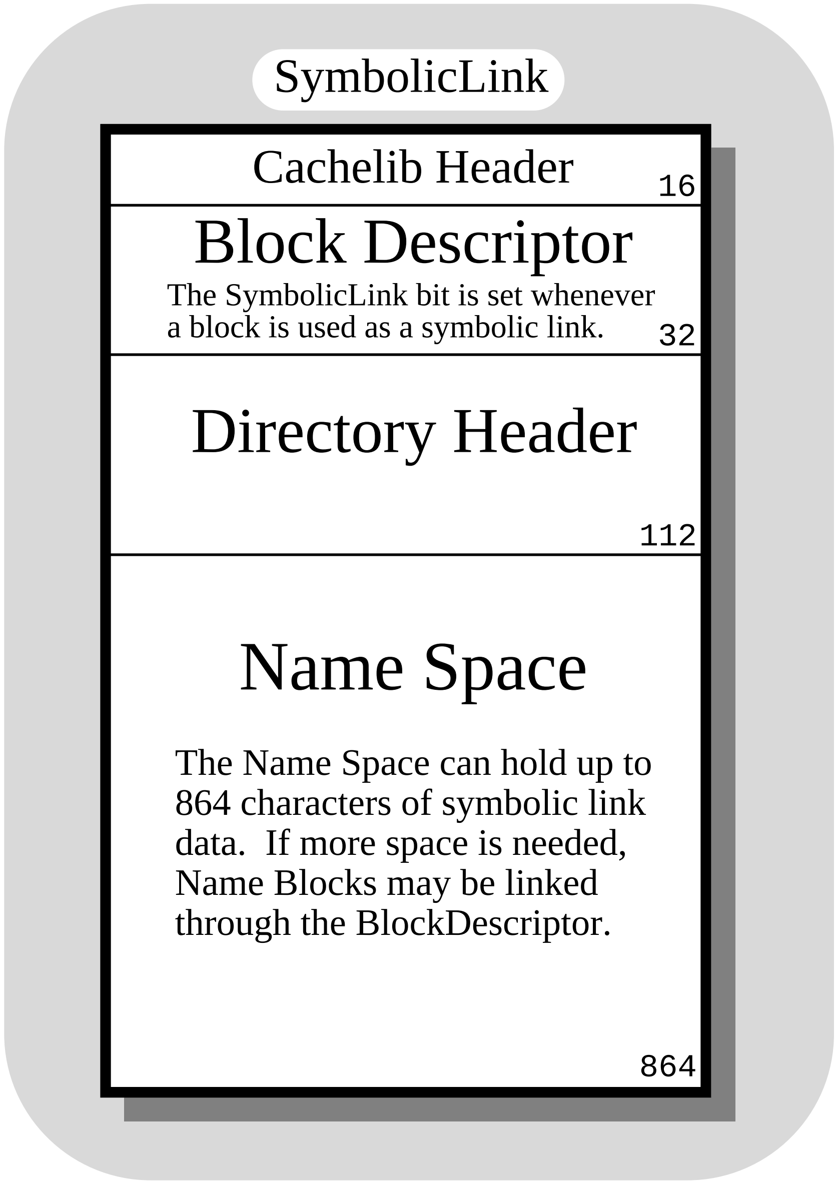
SymbolicLink
Cachelib Header
16
Block Descriptor
The SymbolicLink bit is set whenever a block is used as a symbolic link.
32
Directory Header
112
Name Space
The Name Space can hold up to 864 characters of symbolic link data. If more space is needed, Name Blocks may be linked through the BlockDescriptor.
864

Figure 10. An overview of a Symbolic Link block. The field widths in this diagram are given in units of bytes.

In include file `nsdir.h` a NameBlock is defined as:

```
typedef struct
{
    BlockDescriptorStruct  BlockDes;
    DirectoryHeaderStruct  DirHdr;
    SymbolicLinkName       Name;
} SymbolicLinkStruct;
```

The CachelibHeader is actually part of the structure of the block obtained from the cachelib software, so its structure is not defined as part of the SymbolicLink structure.

2.0 Shiva

The Name Server is a multi-threaded process that manages the UniTree directory structure. Most of the Name Server threads are dedicated to servicing requests made by clients. However, one of the Name Server threads, called *Shiva*, after the Hindu goddess of destruction, is dedicated to removing expired objects from ".trash" directories.

2.1 Trash Directories

A ".trash" directory is created in each user's UniTree home directory when the user is added to the UniTree system. The purpose of the ".trash" directory is to provide a temporary holding place from which a user might recover files that have been inadvertently deleted. When a user deletes a file, it is moved to the ".trash" directory. Any file that has been in the ".trash" directory longer than the expiration time is eligible for complete destruction by Shiva.

2.2 Expiration Time

The Name Server permits each ".trash" directory to have an expiration time associated with it. This expiration time can be set when the directory is created, or may be changed at a later time. In addition, the Name Server has a site configurable default expiration time that is used for ".trash" directories that do not have a specific expiration time. The global default expiration time can be queried and modified with system administrator commands. At any time, a ".trash" directory with a specific expiration time can be changed to use the global default expiration time. (To change or query the expiration time of a specific directory or the Name Server's default expiration time, commands in the utility, `Bonnie`, may be used.)

2.3 Deleting Expired Entries

Shiva awakens periodically to rumble through each ".trash" directory in the UniTree system. The amount of time that Shiva suspends between the trash rummaging and the maximum number of files to remove at a time are site configurable. (A command in `Bonnie` may also be used to change the suspend time and the maximum number of entries to delete at a time.)

Shiva maintains an internal table that maps a UID to a ".trash" directory. When Shiva awakens, she begins with the first entry in this table. Shiva gets a list of the first *n* entries in the ".trash" directory. The entry names are encoded with the time and date they were moved into the ".trash" directory. That time is subtracted from the current time and if that value is greater than the expiration time, the entry is deleted from the ".trash" directory. If not, the entry is left in the trash, and Shiva moves on to the next entry in the list. Shiva advances to the next entry in the UID to ".trash" table when all *n* entries have been examined. If a ".trash" directory has more than n entries, several passes through Shiva may be required before all of the files are removed. After Shiva has rumbled through every entry in the UID to ".trash" table, her insatiable appetite will be somewhat abated, and she will once again go to sleep.

2.4 UID to Trash Directory Mapping

To quickly locate a user's ".trash" directory the Name Server builds and maintains a table called the TrashHash table that maps UIDs to trash can Ids. Trash can Ids are the Cachelib block identifier of the ".trash" directory's BlockHead. In addition to the use described above in section 2.3, the TrashHash table is used by the Name Server whenever a directory entry containing a file capability is deleted and is to be moved into a ".trash" directory.

The TrashHash table is constructed each time the Name Server is initialized. The Name Server builds the table from information found in the password file. Each entry in the password file contains a user's UID and a path name to the user's home directory. Since ".trash" directories reside in the user's home directory, the trash can Id of each user's ".trash" directory can be found and entered into the TrashHash table.

A simple hash is performed on the UID and the resulting value serves as an index into the TrashHash table's array of pointers. Each pointer is the beginning of a linked list of data structures which map UIDs to trash can Ids. Figure 11 illustrates the structure of the TrashHash table.

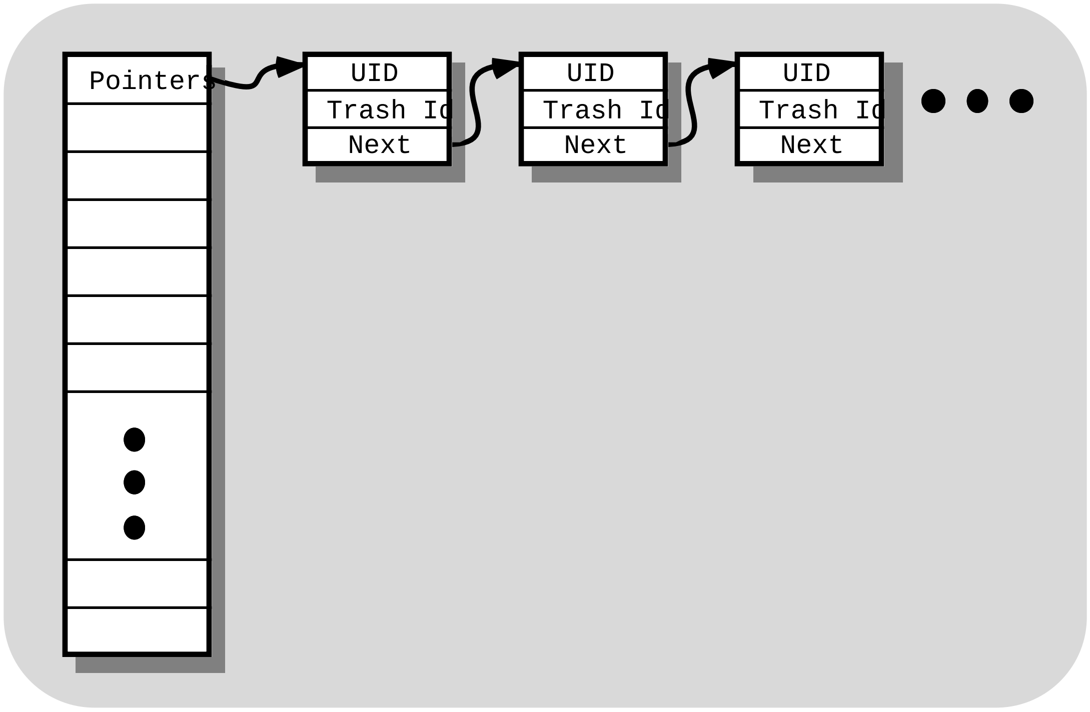

Figure 11. The TrashHash table which maps UIDs to trash can Ids.

In include file `nsdir.h` a the TrashHash table is defined as:

```
typedef struct TrashHashTable
{
    Semaphore         Sem;
    TrashHashElement  *THE[NUM_LISTS_IN_TRASH_HASH];
} TrashHashTable;
```

and the elements in the array of lists are defined as:

```
typedef struct TrashHashElement
{
   struct TrashHashElement  *Next;
   int                       UID;
   unsigned                  TID;
} TrashHashElement;
```

2.5 Names of Items in Trash Directories

Whenever Entries are put into a ".trash" directory they are given a name that encodes the time they were put into the ".trash" directory to make them unique, yet still easily identifiable. The ".trash" entry name is constructed by appending the Date, Time, and a “uniquifier” to the original name. The format of a trash can name is:

```
<OriginalName>#<Date>#<Time>#<Uniquifier>
```

where

`Date`	is in the format mm-dd-yy
`Time`	is in the format hh:mm:ss
`Uniquifier`	is a monotonically increasing 4 digit number.

So, if on November 2, 1993 at 3:46 PM a user deletes a file entry that was named

```
ThisReallyCoolFile
```

and the Uniquifier value is currently 1233, the ".trash" directory entry for this file would be named

```
ThisReallyCoolFile#11-02-93#15:46:00#1234
```

3.0 Software levels

The Name Server software can be divided into seven distinct layers. Figure 12 depicts these layers. Notice that each of these layers depend on the SMILE tasking layer.

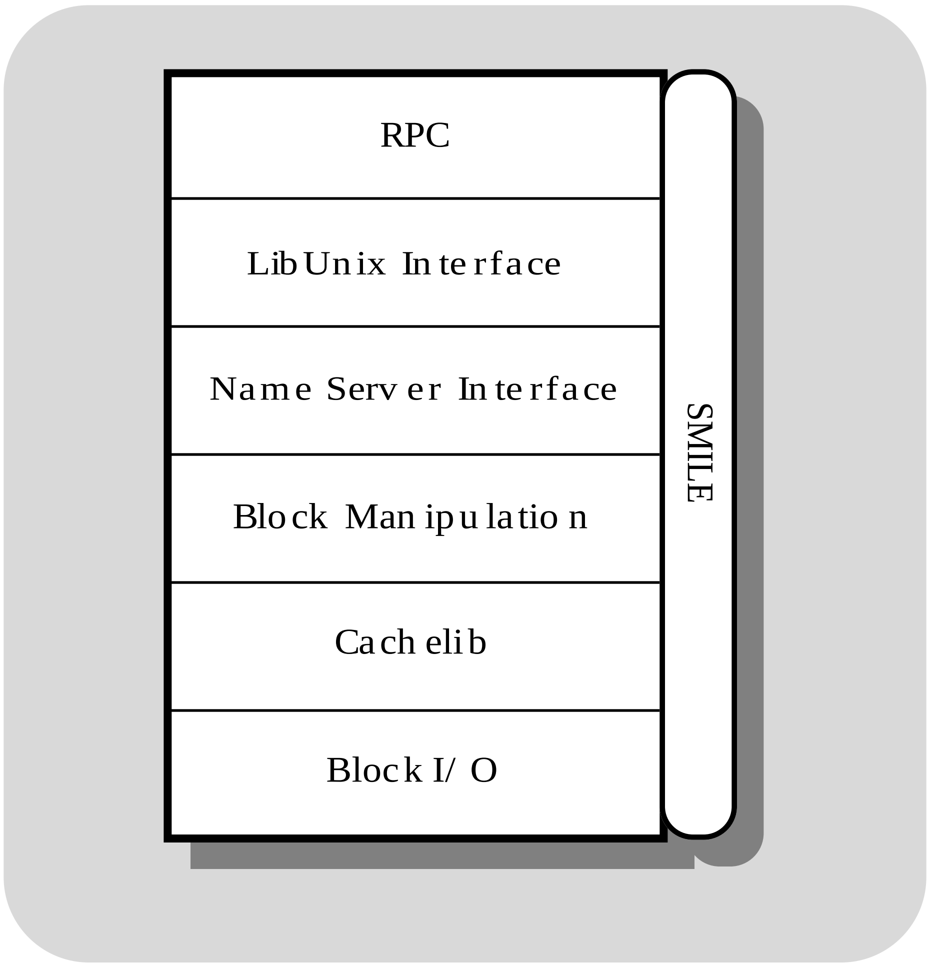

Figure 12. The seven software layers that comprise the UniTree Name Server and the SMILE tasking layer.

3.1 RPC Layer

The RPC layer is composed of the procedures that implement the Remote Procedure Call interface. These routines provide the communication mechanism through which the Name Server communicates with client processes. In addition, the RPC layer implements the "thread" package which allows the Name Server to run as a multi-threaded process.

The user level routines in the RPC layer are divided into two distinct categories: the *client* routines and the (Name) *server* routines. The client routines are found in a library called "libunix.a". For a complete discussion of the client routines refer the libunix.a documentation. The server routines are "called" by the server side of the RPC layer. The routines called by the server side of the RPC layer compose the LibUnix software layer shown in Figure 12.

3.2 LibUnix Layer

The name "LibUnix" is a remnant of by-gone times and even in those times was a poorly chosen name. However, as with most historical mistakes, it is hard to get rid of and therefore remains. The so-called LibUnix routines in the Name Server are all confined to a single file named “LFuncs.c”. The name “LFuncs” was chosen because all of the LibUnix routines begin with the letter 'l' and all of the routines are value returning functions. The routines in this layer are called by the RPC layer.

Each routine in the LibUnix layer in the Name Server has a “mirror-image” routine in the client library libunix.a. Figure 13 shows an example in which a client process is attempting to insert a file capability into a directory.

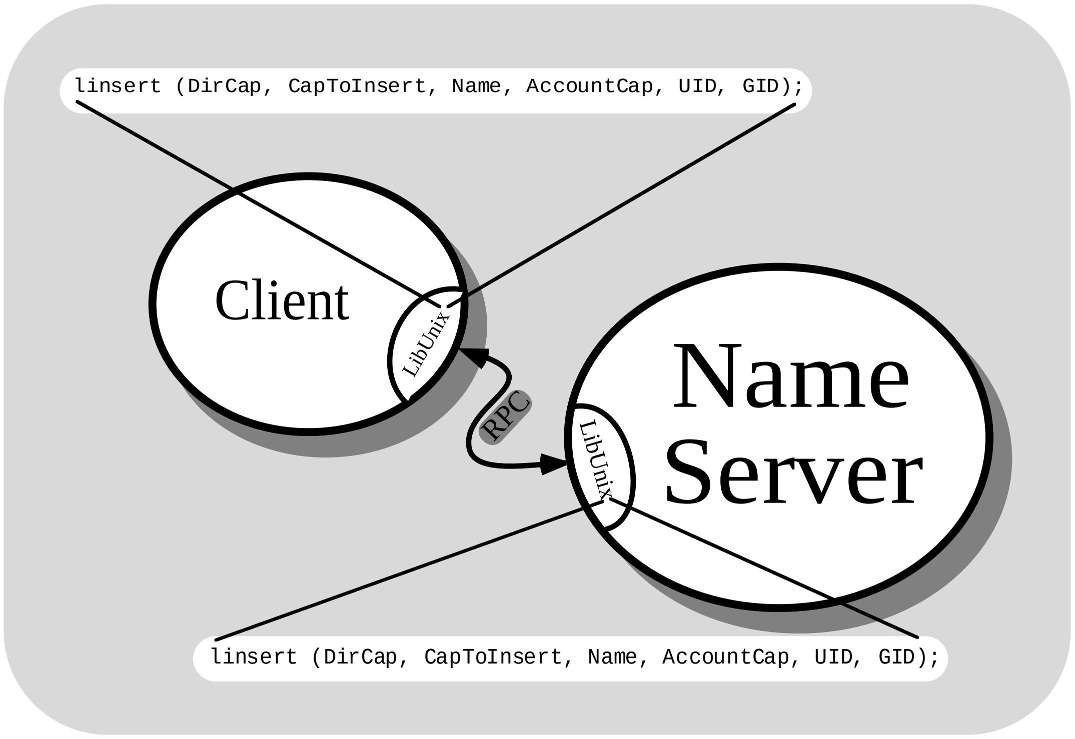
linsert (DirCap, CapToInsert, Name, AccountCap, UID, GID);
Client
LibUnix
RPC
Name Server
LibUnix
linsert (DirCap, CapToInsert, Name, AccountCap, UID, GID);

Figure 13. Notice that an `linsert` procedure exists in both the client LibUnix layer and in the server LibUnix layer.

Although Account capabilities are passed to all of the LibUnix routines, they are, in almost all cases, ignored. They are passed into the LibUnix layer, but are not passed to any of the lower software layers. Account capabilities were used in the original versions of LibUnix and are included in the present version to preserve the function signatures. There is, however, <u>one</u> somewhat ugly instance in which Account capabilities <u>are</u> used. At LLNL there is a need, when making a directory, to pass a *protection level* from the FTP Daemon to the Name Server. Because there are no provisions for a protection level in the `lmakedir` procedure, the protection level is passed by putting it into the protection level field of the Account capability. The Name Server `lmakedir` function extracts the protection level from the Account capability and passes it through to the routine in the Name Server Interface layer. We aren’t proud of this, but we do it. Adding an additional protection level parameter to `lmkdir,` at this time, was considered to be too painful.

The following routines comprise the LibUnix layer

lchange

To change fields of a directory header. In olden times when LINCS was in full flower and the earth was indeed a beautiful place the `lchange` function was used to change any and all of the directory header fields. In these harsh and misbegotten times the only fields that remain to be changed are the ExpirationTime and the ProtectionLevel. And time advances ever forward.

```
    int
lchange ( DirCap, Account, Label, Data, UID, GID )
    Capability          *DirCap;
    Capability          *Account;
    int                  Label;
    int                 *Data;
    int                  UID;
    int                  GID;
```

lcrt

To create a directory and return its capability.

```
    int
lcrt ( Addr, ProtectionLevel, Account, UID, GID, Perms, DirCap )
    NetAddress          *Addr;
    unsigned             ProtectionLevel;
    Capability          *Account;
    int                  UID;
    int                  GID;
    int                  Perms;
    Capability          *DirCap;
```

ldelete

To delete an entry specified by name from a directory.

```
    int
ldelete ( DirCap, Name, Account, UID, GID )
    Capability          *DirCap;
    char                *Name;
    Capability          *Account;
    int                  UID;
    int                  GID;
```

ldestroy

To destroy a block from the Name Server database file. Notice that “destroy” is a relative term—the block is being destroyed from the Name Server’s point of view, but is only being deleted from Cachelib’s point of view.

```
    int
ldestroy ( Id )
    unsigned            Id;
```

lfetch

To fetch the capability specified by name from a directory.

```
    int
lfetch ( DirCap, Name, Account, UID, GID, ReturnedCap )
    Capability            *DirCap;
    char                  *Name;
    Capability            *Account;
    int                    UID;
    int                    GID;
    Capability            *ReturnedCap;
```

lgetattr

To return the attributes of a directory through a UNIX stat structure.

```
    int
lgetattr ( DirCap, StatBuf, Account, UID, GID )
    Capability            *DirCap;
    struct stat           *StatBuf;
    Capability            *Account;
    int                    UID;
    int                    GID;
```

lgetnsparams

To get (fetch) the Name Server global parameters.

```
    int
lgetnsparams ( SysAdmin, UID, GID )
    NSSysAdminStruct      *SysAdmin;
    int                    UID;
    int                    GID;
```

lgetstats

To get the statistics kept by the Name Server.

```
    int
lgetstats ( StatRec, UID, GID )
    NSCompleteStatStruct  *StatRec;
    int                    UID;
    int                    GID;
```

lgettrash

To get the expiration time of a ".trash" directory. Since lgetattr uses the UNIX stat structure to return the attributes and it has no field that maps to the ExpirationTime, a specialized function is needed.

```
    int
lgettrash ( DirCap, Account, UID, GID, ExpirationTime )
    Capability            *DirCap;
    Capability            *Account;
    int                     UID;
    int                     GID;
    int                    *ExpirationTime;
```

linsert

To insert a capability into a directory with an entry name of Name.

```
    int
linsert ( DirCap, CapToInsert, Name, Account, UID, GID )
    Capability             *DirCap;
    Capability             *CapToInsert;
    char                   *Name;
    Capability             *Account;
    int                     UID;
    int                     GID;
```

linsertnolink

To insert a capability into a directory with an entry name of Name. `linsertnolink` differs from `linsert` in that the link count of the item being inserted is not incremented. It is assumed the caller has already taken care of the link count. This routine was written to satisfy an efficiency requirement of saving an RPC message when newly created files are inserted into a directory.

```
    int
linsertnolink ( DirCap, CapToInsert, Name, Account, UID, GID )
    Capability             *DirCap;
    Capability             *CapToInsert;
    char                   *Name;
    Capability             *Account;
    int                     UID;
    int                     GID;
```

lmakedir

To create a directory called Name in the directory specified by DirCap. The protection level is passed through the Account capability. See the discussion at the top of this section.

```
    int
lmakedir ( DirCap, Name, Perms, Account, UID, GID )
    Capability             *DirCap;
    char                   *Name;
    int                     Perms;
    Capability             *Account;
    int                     UID;
    int                     GID;
```

lmvdirto

To move an entry named OldName from directory OldDirCap to new directory NewDirCap with the name NewName.

```
   int
lmvdirto ( OldName, NewName, OldDirCap, NewDirCap, Account,
           UID, GID )
   char                 *OldName;
   char                 *NewName;
   Capability           *OldDirCap;
   Capability           *NewDirCap;
   Capability           *Account;
   int                   UID;
   int                   GID;
```

lquiesce

To “quiesce” the Name Server. That is, to have the Name Server flush any modified objects from the cache and enter ReadOnly state (where directories may be read, but not written). This means that the TimeLastAccessed field in the Directory Header *will not* be updated.

```
   int
lquiesce ( UID, GID )
   int                   UID;
   int                   GID;
```

lrddir

To read the specified number of entries from a directory and return the entry names, capabilities, and associated cookie values. The strategy in this routine is to read the desired stuff from the directory using an “NS” routine and an efficient Name Server data structure. The data is then copied into the traditional NFS specified format.

```
   int
lrddir ( DirCap, StartIndex, Count, Names, Caps, Cookies,
         Account, UID, GID )
   Capability           *DirCap;
   int                  *StartIndex;
   int                  *Count;
   char                 *Names[];
   Capability            Caps[];
   char                 *Cookies[];
   Capability           *Account;
   int                   UID;
   int                   GID;
```

lrdlink

To return the contents of a symbolic link.

```
   int
lrdlink ( SymCap, Account, UID, GID, Contents )
   Capability           *SymCap;
   Capability           *Account;
   int                   UID;
   int                   GID;
   char                 *Contents;
```

lrenameto

To rename an entry in a directory, changing the entry's name from OldName to NewName.

```
    int
lrenameto ( OldName, NewName, DirCap, Account, UID, GID )
    char                  *OldName;
    char                  *NewName;
    Capability            *DirCap;
    Capability            *Account;
    int                    UID;
    int                    GID;
```

lreplace

To replace the capability in a directory entry specified by Name with a new capability.

```
    int
lreplace ( DirCap, ReplacingCap, Name, Account, UID, GID )
    Capability            *DirCap;
    Capability            *ReplacingCap;
    char                  *Name;
    Capability            *Account;
    int                    UID;
    int                    GID;
```

lrestart

To change the Name Server’s ReadWrite mode to Write mode after the Name Server has been “quiesced.” `lrestart` re-starts normal operation.

```
    int
lrestart ( UID, GID )
    int                    UID;
    int                    GID;
```

lsetattr

To set whatever attributes the caller wishes to have set in the directory header. There is an interface change that takes place here because the caller supplies their attributes in a UNIX stat structure and `lsetattr` puts these attributes into an AttributesStruct for presentation to the Name Server.

```
    int
lsetattr ( DirCap, StatBuf, Account, UID, GID )
    Capability            *DirCap;
    struct stat           *StatBuf;
    Capability            *Account;
    int                    UID;
    int                    GID;
```

lsetnsparams

To set one or more of the Name Server global parameters.

```
    int
lsetnsparams ( SysAdminRec, UID, GID )
    NSSysAdminStruct      *SysAdminRec;
    int                     UID;
    int                     GID;
```

lsettrash

To set the expiration time of a ".trash" directory. Once again we find ourselves implementing something for no other reason than to maintain compatibility with the mistaken past. Why not simply use the `lchange` function since one of its major purposes and reasons of existence is to change the expiration time? Why not indeed.

```
    int
lsettrash ( DirCap, Account, UID, GID, ExpirationTime )
    Capability            *DirCap;
    Capability            *Account;
    int                    UID;
    int                    GID;
    int                    ExpirationTime;
```

lsymlink

To create a symbolic link with the specified contents in a directory with an entry name specified by Name.

```
    int
lsymlink ( DirCap, Name, Account, UID, GID, Permissions,
           Contents )
    Capability            *DirCap;
    char                  *Name;
    Capability            *Account;
    int                    UID;
    int                    GID;
    int                    Permissions;
    char                  *Contents;
```

3.3 Name Server Interface Layer

This layer contains all of the procedures that make up the “natural” interface to the Name Server. These are the procedures that perform the individual Name Server functions. There is, in fact, a Name Server Interface procedure for each Name Server function. For example, the procedure `NSMakeDir` performs the function that makes a directory. In a perfect world the Name Server Interface procedures would replace the LibUnix layer. However the world is sometimes imperfect and, because the LibUnix interface procedures were here first, they remain.

The following routines comprise the Name Server Interface layer:

```
NSCreate
NSDelete
NSFetch
NSGetAttribs
NSInsert
NSMakeDir
NSMakeSymLnk
NSMoveDir
NSReadDir
NSReadSymLnk
NSSetAttribs
NSSysAdmin
```

3.3 Block Manipulation Layer

The Block Manipulation software layer contains all of the procedures that perform the utilitarian functions performed by the Name Server. There are procedures that set bits in bit arrays, that add name-capability pairs to directories, that remove name-capability pairs from directories, that list the contents of directories, etc.

Library package

There is a small collection of procedures that are not specific to the Name Server, and in fact have a somewhat general application. Some of these procedures are called by other processes such as `NSDE` and `Bonnie`. These procedures have been collected in a file named `Library.c` (it seems as though they should be in a library somewhere). The names of these “library” routines are:

```
ConvertTime
DayOfWeek
GetSymbol
IsDST
```

GetName package

The two procedures `GetName` and `GetSymName` are found in the file `GetName.c`. `GetName` is used to extract long (more than 16 characters) names from a directory Entry and `GetSymName` is used to extract the symbolic link data from the Name field of a SymbolicLink.

`ListDir` calls `GetName` whenever it needs to return an Indirect Entry name. `GetName` scans through all of the (needed) NameBlocks totaling up the length of the Indirect name. Once the total length of the name has been determined heap space is acquired and all of the various pieces of the name are copied into this space making the name one contiguous string. `GetName` then returns a pointer to this string.

`GetSymName` is called by `NSGetSymLink` to read SymbolicLink data.

FindEntry package

The Name Server often has occasion to find an entry given its name. Such occasions arise when inserting and deleting entries into and from directories.

To simplify and isolate the somewhat difficult task of finding an Entry, the procedure `FindEntry` is provided. `FindEntry` and all of its associated parts are found in the file `FindEntry.c`. `FindEntry` is supplied a pointer to a BlockHead and a Name to find. The search begins in the BlockHead and, if necessary, continues through any attached EntryBlocks until the name is found or all the entries in the directory have been examined. The name of the entry to find can be arbitrarily long. `FindEntry` will fetch whatever NameBlocks are needed to perform its search. If a match to a name is found `FindEntry` returns the associated capability and returns a boolean value of “true”.

AddEntry package

All of the procedures needed to add an Entry (a name-capability pair) to a directory are isolated in the source file `AddEntry.c`. `AddEntry` is given a Name and a capability and the directory to add them to. `AddEntry` finds the first available entry slot and inserts the entry there. If the name contains 16 characters or less a DirectEntry is made. If the name contains more than 16 characters an IndirectEntry is made. If an additional EntryBlock is needed, `AddEntry` obtains one. If additional NameBlocks are required, `AddEntry` obtains them. `AddEntry` updates the InUse, Indirect, and NameSpaceUsage bits as appropriate.

RemoveEntry package

All of the procedures needed to remove an Entry (a name-capability pair) are isolated in the source file `RemoveEntry.c`. `RemoveEntry` is given a Name and the directory to remove the entry from. `RemoveEntry` locates the name (if it exists) and removes the Name and the corresponding capability from the directory. If the name is 16 characters or less (a DirectEntry) the DirectEntry is removed by simply zeroing the 96 bytes. If, however, the name contains more than 16 characters (an IndirectEntry) the 96 bytes are zeroed and the characters comprising the remainder of this name are removed from the NameBlocks.

If the removal of characters from a NameBlock causes a NameBlock to become empty, `RemoveEntry` scavenges the NameBlock. If the removal of an Entry causes an EntryBlock to become empty, `RemoveEntry` scavenges the EntryBlock *if* it is the last EntryBlock in a chain. The reason for this is that scavenging free EntryBlocks in the middle of a chain would change the “Cookie” index of the subsequent Entries. However, if there are one or more empty EntryBlocks chained together and the last EntryBlock contains a single Entry that is being removed, then `RemoveEntry` will scavenge all of the empty EntryBlocks.

Listing a directory.

Listing the contents of a directory is accomplished by calling ListDir which is found in file ListDir.c. The caller supplies the directory to be listed, where to start listing in the directory, and a count of how many Entries are to be listed. The Entries are returned in a linked list of structures called a DirIdListStuct.

Algorithm for detecting potential loops

The directory structure provided by the Name Server is a tree structured directory. A tree structure by definition cannot contain "loops". Because clients can insert directory capabilities and can move directories the Name Server needed some method of insuring that inserts and moves would not create loops in the directory structure. There is a procedure named `CheckForLoop` in file `CheckForLoop.c` that performs this check. The input to `CheckForLoop` is the capability to the directory that is being inserted into and the capability to the directory that is to be inserted.

The algorithm for checking for a loop is straight forward: Starting with the directory that is being inserted (or moved) into, the ".." (dot-dot) directory is examined. If the ".." capability is equal to the capability that is to be inserted, there is a loop. Following the path created by the ".." directories, `CheckForLoop` continues testing for a match against the capability that is to be inserted. If the capability to be inserted matches any of the ".." capabilities, there is a loop. `CheckForLoop` finishes its search when the ".." directory capability equals the “.” directory capability, because this only occurs in the top level directory (the RootOfRoots).

3.4 Cachelib Layer

Cachelib is a collection of software that allows the Name Server to manage its data base. Cachelib maintains the data as an array of fixed sized elements called objects. The objects are stored in a file called the store file. A store file may consist of several physical files, but cachelib treats the files as a single logical file. Each object in the store file has a unique cachelib identifier, which is a zero origin index into the object array.

In addition, Cachelib provides the Name Server with:

- an in-memory cache
- access synchronization in a multi-tasking environment
- data integrity through the use of primary and backup files
- a transaction mechanism for ensuring consistency of multiple block updates

3.4.1 In-memory Cache

An in-memory cache is obtained from heap space by `occinit`. The size of the cache (number of objects that can be held) is a site-configurable parameter that can be changed in the configuration file and picked up next time the Name Server is initialized. The cache is implemented as an array of elements called slots. Each slot contains an object's data (read from the store file) and some state information about the object, such as the cachelib id of the object and whether the slot is dirty or not.

When the Name Server references an object, the Cachelib layer determines if the object is in the cache or not. If the object is not in the cache, the Least Recently Used (LRU) cache slot is found and the object is read from the store file into that slot. Cachelib keeps a linked list of LRU slots where the slots at the beginning of the list are chosen first when searching for an available slot. All modified objects are marked as dirty and are written back to the store file before being removed from the cache.

3.4.2 Access Synchronization

Cachelib provides a flexible mechanism for synchronizing access to objects in a multitasking environment. It provides this synchronization through two user level procedures that manipulate internal semaphores. Name Server tasks obtain access to objects by calling `oclock` and specifying the Id of the desired object. Upon receiving the `oclock` request Cachelib determines whether or not the object is already in the memory cache. If the object is not in the memory cache Cachelib finds an available slot and reads it in. If the object is in the cache Cachelib simply P's the slot semaphore. Access to the slot is strictly first-come first-served.

A second parameter to `oclock` determines the type of access: shared or exclusive. Tasks requesting a shared lock share access to a slot with other tasks. Tasks having a shared lock should not write to or alter the data in the slot. (There is no way Cachelib can enforce this requirement. This is strictly a coding discipline.) Tasks requesting exclusive access to a slot wait until all shared locks have been given up and then exclusive access to the slot is granted. Once an exclusive lock is granted, any other requests (shared or exclusive) for the slot must wait until the exclusive lock is released. The holder of an exclusive lock is free to modify the object.

Access to slots (objects) is released through the `ocunlock` procedure. One of the parameters to `ocunlock` is the Id of the block that is to be released. Upon receiving an `ocunlock` request Cachelib V's the indicated slot semaphore thereby making the slot available to another task.

Sometimes, when examining Name Server core files it is helpful to know the last few events that occurred regarding slot access. For this reason, a pointer to a history buffer exists in the cache data structure. The history buffer contains a list of records defined in `cache.h` as:

```
typedef struct
{
    unsigned   TaskId;
    unsigned   IdRequested;
    unsigned   IdFromSlot;
    unsigned   NextIdFromSlot;
    unsigned   SlotAddress;
    unsigned   State;
    unsigned   TheTime;
} HistStruct;
```

Where `State` an integer value defined in `cache.h` by:

```
typedef enum
{
    STATE_ID_PRESENT;
    STATE_ID_BEING_WRITTEN;
    STATE_ID_BEING_READ;
    STATE_ID_BEING_FREED;
    STATE_SLOT_CLEAN;
    STATE_SLOT_DIRTY;
    STATE_DONE;
    STATE_RETRY;
    STATE_ERROR;
} States;
```

In the cachelib routine, `slot.c`, a task transitions through some of these states before obtaining access to the desired object.

3.4.3 Data Integrity

Cachelib provides data integrity through the use of two devices (or files): a primary and a backup device. Cachelib is very insistent on the following point: there must be a distinct primary and backup device. *In order for the transaction mechanism to operate correctly* ***there must be two distinct devices***.

The data on the backup device should be an exact copy of the data on the primary device. The only time the data ever differs is following a hard crash that has left the database in an inconsistent state. The differences should be immediately resolved after the Name Server initializes. Differences that remain following Name Server initialization are matters of grave concern and should be immediately examined and rectified by the system administrator.

All device reads are from the primary device unless an I/O failure occurs in which case the backup device will be read. If Cachelib is unable to read from either device it halts and demands that the disks be repaired. All writes are to both the primary and the backup device. The transaction mechanism dictates that all objects in a transaction are first written to the primary device and then these same objects are written to the backup device.

3.4.4 Cachelib Transactions

Cachelib offers data consistency by allowing multiple blocks updates to occur as a single atomic transaction. At the Cachelib level there is no way to know how or if objects are related. Cachelib simply moves single (1024-byte blocks) objects between the memory cache and the store file(s). It is the responsibility of the higher level software to bind objects into logical groupings. The Name Server is the higher level software that makes these groupings.

Whenever directory entries are added or deleted it can be the case that two or more blocks are altered (created, deleted, updated). For example, if an Entry with a very long name is added, one or more NameBlocks may be obtained and an EntryBlock may be updated or obtained. Likewise, if an Entry is deleted, several NameBlocks and/or EntryBlocks may be altered. It is vitally important that all of these altered blocks are updated as a single transaction and if a failure of any sort occurs, nothing should be updated. If any portion of the update is interrupted it should look to the user as if the command never reached the Name Server.

The Name Server groups blocks together with a linked list data structure called a TransEntryStruct. As blocks are updated and changed they are added to the linked list of TransEntryStructs. When it is time to write these blocks to disk the Name Server calls the Cachelib procedure WriteSlots passing the list of

TransEntryStructs. WriteSlots uses a “stable store” algorithm to insure the blocks are written to the store files as an atomic transaction.

The stable store algorithm operates by writing its “intentions” to a file called a “Transaction Log File”. These intentions take the form of “all the cachelib Ids that are to be written as a group.” For example, if WriteSlots is given the Ids of four blocks that are to be written from the memory cache to the store files, it first writes these four Ids into the Transaction Log File. The format of a Transaction Log File is shown in Figure 14. After the Transaction Log File is safely written, WriteSlots then writes the individual blocks to their respective slots. First it writes to all the objects in the primary file and then to these same objects in the backup store file. This simple algorithm insures that these blocks are written as a single transaction because if a failure of any sort (Kernel panic, hardware fault, etc.) occurs, the next time the Name Server is initialized, the Cachelib initialization software examines the Transaction Log File and corrects any inconsistencies that might have occurred. The information in the Transaction Log File is sufficient to resolve any inconsistencies that may have occurred at any point in the transaction.

There is an interesting problem that may occur. What happens if the system (repeatedly) crashes during Cachelib initialization? The stable store algorithm must recognize this situation and survive it. To solve this potential problem a second file called the “Transaction Journal File” is used. This file is used only during initialization to record the progress of any restore efforts. Using the Transaction Log File and the Transaction Journal File, Cachelib can insure consistent multi-block updates.

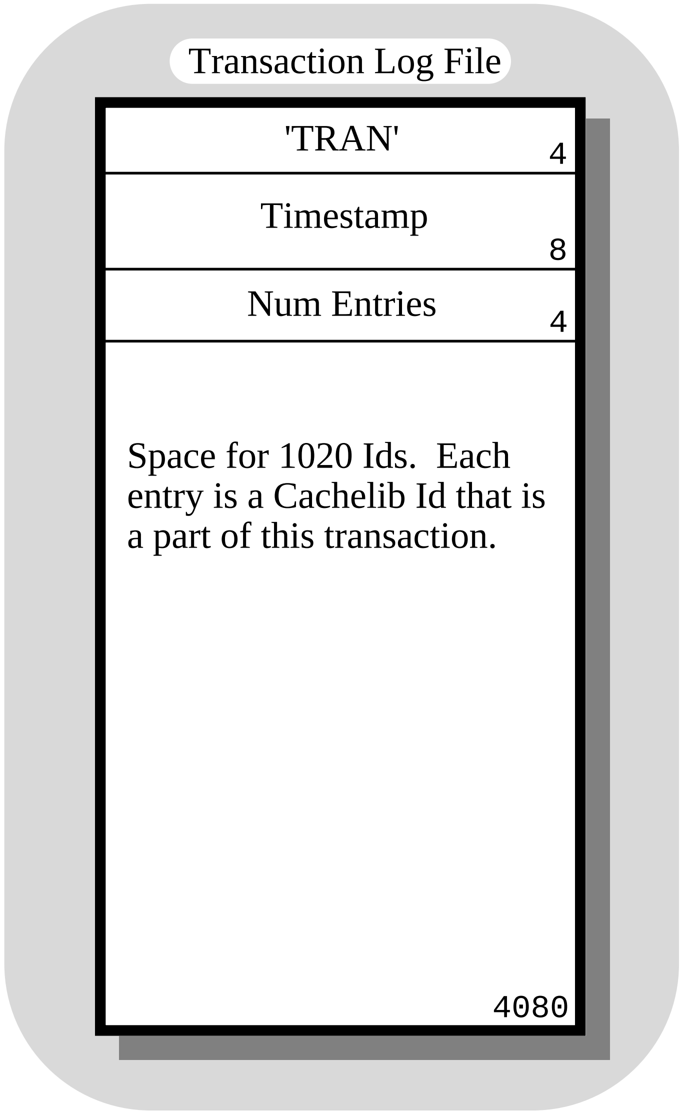
Transaction Log File
'TRAN'
4
Timestamp
8
Num Entries
4
Space for 1020 Ids. Each entry is a Cachelib Id that is a part of this transaction.
4080

Figure 14. The Transaction Log File is composed of 4 fields: a consistency string "TRAN", the current time, the NumEntries (the number of block Ids to be found in the Id array), the Ids of the blocks to be written during this transaction.

In include file `cache.h` the Transaction Log File is defined as:

```
typedef struct TransJournalStruct
{
   char  ConsistencyString[4];
   int   NumEntries;
   int64 TimeStamp;
   int   Entry[MAX_TRANS_ENTRIES];
} TransJournalStruct;
```

3.5 Block I/O Layer

The Block I/O layer is used by Cachelib to perform its disk I/O. This layer is necessary to perform “device” (also called “partition”) I/O because I/O to devices can only be done in 4096 byte blocks and the I/O must begin and end on 4096 byte boundaries. However, Cachelib wants to read and write objects that are less than this. Each Name Server block is 1024 bytes in length. The Block I/O layer provides a solution to this problem by providing a cache of several 4096 byte buffers. When Cachelib reads one of its (small) objects, the Block I/O layer finds an empty (large) buffer from among its pool of buffers and reads in the 4096 bytes that contain the object. The object is then copied into the requester's buffer. If an empty buffer cannot be found, the Block I/O layer obtains a new one.

When the files used by the Block I/O routines to read and write the disk are opened an “option" that insures that the data will not pass through any UNIX buffers is used. It is vitally important to the correct operation of the Name Server that all writes to a device are exactly that, a write to a device, not to some system buffer.

Overlapping, or asynchronous I/O, is accomplished through the tasking mechanism. While a task is waiting for its I/O to complete, another task may run and “launch” its I/O. As each I/O completes the corresponding task is awakened.

3.6 SMILE

SMILE (for **S**ystem/**M**achine-**I**ndependent **L**ocal **E**nvironment) is a portable tasking package developed at LLNL. The Name Server relies on this software layer for its multi-tasking environment. For more information on SMILE see the SMILE documentation.

4.0 Required Files

There are several files that must be present for the Name Server to operate correctly. Some of these files are configuration files which contain run-time configuration parameters, and others are data files used by the Name Server. The path names to all of these files are obtained through calls to `ResolvePath`.

Configuration files (read at initialization):

- Name Server configuration file.
- file containing the RootOfRoots capability.
- password file.

Data files used by the Name Server:

- primary database file.
- backup database file.
- transaction log file.
- transaction journal file.
- Name Server log file.
- Cachelib log file.

5.0 Initialization

When the Name Server "comes up" (initializes) it goes through a number of steps:

- It initializes two of its log files: NSLog and NSCacheLog. The Name Server writes the current time and the values of all the initialization parameters into the NSLog file. Cachelib writes to the NSCacheLog file.
- It discovers its own address by a call to `ResolveNetAddr`.
- It reads a number of global parameters from the configuration file. These parameters tell the Name Server the size of its memory cache, the number of store files, the types (file or device) of the store files, and whether or not to do checksumming.
- It initializes (zeros) its statistics records.
- It initializes the storefiles with a call to the Cachelib routine `osfinit`. During this step all repairs (if needed) to the store files are performed. If there was a crash that left the store file(s) in an inconsistent state Cachelib, using the transaction mechanisms, will correct the inconsistencies.
- It builds and initializes its memory cache with a call to the Cachelib routine `occinit`.
- It builds the TrashHash table used by Shiva.
- It forks the Shiva task.
- It forks several tasks that service client requests.

6.0 Core files

If the Name Server detects an internal inconsistency it will "crash". A crash is always done in two steps: it writes a message to the NSLog and then calls the `CRASH` macro. The `CRASH` macro calls the `Crash` procedure passing it the name of the source file containing the currently executing procedure and the current line number. `Crash` writes this information into NSLog. `Crash` then looks for an existing core file in the current working directory and if one is found it changes its name to

```
core-mmdd-hhmn
```

where

`mm`	the month
`dd`	the day
`hh`	the hour
`mn`	the minute

`Crash` then calls `abort` and the Name Server goes to visit its ancestors.

7.0 Orphans

It is possible, if a crash occurs at precisely the wrong time, for the Name Server to create an orphaned block. An orphaned block is a block that is not on the Cachelib free list, and is not pointed to by any other (Name Server) blocks. Orphaned blocks do not cause any trouble for the Name Server and do not in any way cause the database to be inconsistent. In fact, the only problem caused by orphaned blocks is that the space they occupy in the database could be used for better purposes.

An orphaned block can occur in the following 2 ways:

Suppose the last Entry from an EntryBlock is being deleted and that this EntryBlock is the last (or only) EntryBlock in a chain. Recall that whenever the last Entry is removed from the last EntryBlock the block is scavenged. If an inopportune crash occurs during this “update” an orphaned block could result.

Suppose that a new block is obtained because of a lmkdir request or for use as a NameBlock or an EntryBlock. If an inopportune crash occurs during this operation an orphaned block could result.

7.1 How?

To preserve the integrity of the database, deletes are processed as follows:

- all blocks (excluding the “last block”) that have been modified are written as a transaction to the database.
- the “last block” is deleted.

If a crash occurred between these two steps an orphaned block (the “last block” in this example) can occur. Note that orphaned blocks can also occur when attempting to delete the last NameBlock in a chain or when deleting a directory.

To preserve the integrity of the database, new blocks are obtained as follows:

- the cachelib routine `osgeten` is called. `osgeten` gets the next free block in the store file and removes it from the cachelib free list.
- all blocks (those that point to the new block) that have been modified are written as a transaction to the database.

If a crash occurs between these two steps an orphaned block can occur.

7.2 Why?

This problem has to do with history, cost, and time. When transaction processing was put into Cachelib we immediately knew there was going to be a problem with “deletes” and "creates". The transaction mechanism was designed so that the upper level (above Cachelib) software could control the blocks that are included in a transaction. Obtaining and deleting cachelib blocks is a problem because it operates at a level too low to be included in the transaction mechanism and it involved the update of two blocks (and the store file header). Cachelib already had a correct and sophisticated algorithm for detecting and correcting partially allocated blocks. Rather than take the time to redesign the way cachelib obtains and deletes blocks and integrate the new method into the transaction mechanism, we decided to live with the problem and accept the consequences. Since the consequences are that a handful of orphaned blocks may be created each year , who cares? This is a problem that can be lived with for centuries before ever being noticed.

However, even though the problem appears to be trivial, a solution has been provided. `NSDE` has a command that searches a Name Server database for orphaned blocks. If any are found they are reported. After insuring that the reported blocks are indeed orphans, `Bonnie` can be used to zap the blocks (with the `dst` command). It is *strongly* recommended that the "orphan search" done by `NSDE` is performed on a copy of the (non-active) database. It should *never* be performed on the (live) database that the Name Server is currently connected to because the real database can be "caught" in an inconsistent state when new blocks are being added to or removed from directory chains. It is further recommended that "orphan searches" (and other consistency checks) be performed on a routine (monthly) basis.